TwinPCG: Dual Thread Redundancy with Forward Recovery for Preconditioned Conjugate Gradient Methods

TwinPCG: Dual Thread Redundancy with Forward Recovery for Preconditioned Conjugate Gradient Methods

Kiril Dichev
HPDC
School of EEECS
Queen's University Belfast
Email: K.Dichev@qub.ac.uk

Dimitrios S. Nikolopoulos
HPDC
School of EEECS
Queen's University Belfast
Email: D.Nikolopoulos@qub.ac.uk

***Abstract*—Even though iterative solvers like the Preconditioned Conjugate Gradient method (PCG) have been studied for over fifty years, fault tolerance for such solvers has seen much attention in recent years. For iterative solvers, two major reliable strategies of recovery exist: checkpoint-restart for backward recovery, or some type of redundancy technique for forward recovery. Efficient low-overhead redundancy techniques like algorithm-based fault tolerance for sparse matrix-vector products (SpMxV) have recently been proposed. These techniques add resilience with a good, but limited scope; state-of-the-art techniques correct at most 1 fault within a SpMxV. In this work, we study a more powerful resilience concept, which is redundant multithreading. It offers more generic and stronger recovery guarantees, including any soft faults in PCG iterations (among others covering SpMxV), but also requires more resources. We carefully study this redundancy-efficiency conflict. We propose a fault-tolerant PCG method, called TwinPCG, which introduces very small wall-clock time overhead, and significant advantages in detection and correction strategies. Our method uses Dual Modular Redundancy instead of the more expensive Triple Modular Redundancy (TMR); still, it retains the TMR advantages of fault correction. We describe, implement, and benchmark our iterative solver, and compare it in terms of efficiency and fault tolerance capabilities to state-of-the-art techniques. We find that before multithreading in BLAS, TwinPCG introduces 5-6% runtime overhead compared to reference PCG implementations, and can exploit BLAS multithreading well. In the presence of faults, it reliably performs forward recovery for a range of problems, showing all the strengths of TMR techniques.**

***Index Terms*—Conjugate Gradients, Fault Tolerance, Soft Faults, Redundant Multithreading, Dual Modular Redundancy, BLAS**

I. Introduction and Related Work

Iterative solvers like the Conjugate Gradient (CG) method [1], and its preconditioned variations (PCG), are an important and well studied method to solving large systems of linear equations for positive definite matrices. In the absence of faults, PCG shows excellent practical convergence, even if in theory it is susceptible to numerical inaccuracies.

Soft faults in memory, including silent faults, are expected to increase in computer architectures [2]; the domain of Near-Threshold Computing [3] suggests the fault rates will increase even more. In this domain, it is proposed to use supply voltage near the operational threshold of circuits with the goal of increased energy efficiency. Voltage frequency scaling, however, has been demonstrated to increase soft error rates (e.g. [4]).

Online-ABFT [7]	Partial redundant computation
Algorithmic redundancies [8]	Partial redundant data and computation
ABFT SpMxV [9], [10]	Partial redundant data and computation
TwinPCG	2 x Full redundancy
TMR [11]	3 x Full redundancy

TABLE I: State-of-the-art fault-tolerance techniques for PCG in shared memory (including TwinPCG positioning): Redundancy increases from top to bottom, and so do the forward recovery capabilities.

Transient faults are relevant to PCG methods for multiple reasons: First, due to the very large matrices PCG can handle, combined with potentially many iterations, a transient fault may occur. Second, PCG is in the general case very unstable when such faults occur. It is well studied that its convergence can not be guaranteed even for round-off errors, let alone for transient errors which can affect more significant bits in memory.

In the last five years, there has been a rise in solid research efforts to develop fault-tolerant iterative solvers on the example of (P)CG. In general, fault-tolerant iterative methods follow one of two important directions, or combinations thereof:

- Checkpoint-restart
- A "stronger" type of redundancy:
  - Time/space redundancy within single-threaded execution, e.g. algorithm-based fault tolerance (ABFT) for sparse matrix-vector product (SpMxV)
  - Thread redundancy, e.g. Triple Modular Redundancy (TMR)

Unreliable, and hence more efficient, fault tolerance mechanisms exist, e.g. a self-stabilizing approach [5], or a lossy recovery approach [6]. However, in this work we remain in the realm of reliable recovery mechanisms, which guarantee that PCG methods preserve their convergence.

In this section, we detail and compare the state-of-the-art developments in fault-tolerant shared-memory implementations of PCG. We remark that while we do not focus on distributed memory fault-tolerance PCG techniques, an efficient technique exploiting redundancy in distributed memory for PCG is presented in [12]. We reserve the distributed memory direction for future work.

A summary of related fault-tolerance PCG techniques is given in Table I. We will proceed in increasing level of redundancy, since this is a key aspect of our work as well: Online-ABFT [7] is a single-threaded PCG implementation using checkpoint-restart as fault tolerance approach. It is simple enough to adopt, with no parallelism being introduced in the original work. To detect faults, Online-ABFT monitors PCG invariants like residual levels and orthogonality.

An increasing level of redundancy in work on iterative solvers involves ABFT techniques, which was first successfully used by [13] to detect and correct faults in the matrix-matrix product. The basic idea is to introduce additional checksums, and some additional computation, which may allow for detection and correction of faults. The same idea was recently applied to the sparse matrix-vector product in related work ([9], [10]). These contributions improve the resilience of iterative solvers like PCG by focusing on the underlying sparse matrix-vector product, i.e. implementing ABFT SpMxV. The individual techniques differ in their overhead, and in the capability to recover from faults. The entire mechanism of ABFT SpMxV can be considered an efficient redundancy in a single-threaded execution, which uses some extra space (checksums), and extra time (additional computations). There are strict limits to what existing ABFT SpMxV can do: the related contributions [9], [10] detect up to two faults, and correct up to 1 fault. In addition, the sequential use of redundancy in these approaches, while not significant, always impacts runtime: every additional check is expensive, particularly so for an efficient sparse matrix-vector product. For example, [9] measures a 7.5% overhead for their most efficient approach of detection (but no correction) of faults.

Another resilience approach [8] exploits redundancy for soft faults which have already been detected. Once the operating system detects a fault (e.g. a page fault), the authors use algorithmic redundancies (which need to be specified), and interpolation, to efficiently recompute values and roll forward. Detection and correction require the support of the runtime (in contrast to our solution).

The next step of redundancy is in redundant multithreading, and this is where our main contribution lies. Before we outline the few efforts made in this area so far, we list some important advantages of redundant multithreading, compared to non-redundant or less redundant techniques:

- Compared to rollback recovery which uses no data redundancy, further redundancy offers the possibility of forward recovery, always outperforming the former in the presence of faults
- Compared to less redundant methods like ABFT SpMxV, redundant multithreading is significantly more powerful and generic. It essentially allows to recover from arbitrary faults within a data structure (vectors, matrices) used in a PCG iteration. Redundant multithreading goes beyond any ABFT SpMxV strategy in its capabilities. In fact, existing ABFT SpMxV solution offers limited detection and correction capabilities (up to 1 correction), and they only apply to SpMxV.
- While redundant multithreading always multiplies the used CPU and cache resources compared to less redundant techniques, it *can* be implemented to only marginally increase the total runtime, since multithreading is inherently suitable for parallelization. We demonstrate this experimentally in this work.

A very popular and well-established redundancy technique, which can be implemented in hardware or software, is Triple Modular Redundancy (TMR) (e.g. [14]); in this technique, detection of a fault is trivial, and a correction is performed via majority voting of two threads (hopefully) carrying forward correct data. TMR is the minimal thread redundancy approach recently used for various kernels, including iterative solvers like CG, in the work of [15], [11]. The authors use a holistic compile and runtime system to dynamically spawn redundant threads in certain regions, increasing the number of redundant threads as needed. The assumption is that the runtime is able to detect certain types of faults (like ECC errors), and dynamically spawn redundant threads for fault tolerance.

The issue with thread redundancy, particularly TMR, is that it uses triple CPU and cache resources that could otherwise be used for more efficient computation, and this is also clearly demonstrated in our experimental results. As Kanellakis has ably summarized many years before the advent of many-core systems, “parallel algorithm efficiency implies a minimization of redundancy in the computation, leaving very little room for fault tolerance” [16].

We propose a solution to this problem, by implementing TwinPCG, an original fault tolerant PCG algorithm, in which we use the minimum possible redundant multithreading, Dual Modular Redundancy (DMR). DMR is well known, but the novelty is that our solution is capable of forward recovery, similarly to TMR. We integrate backward and forward recovery, and implement a robust fault tolerance solution in TwinPCG. On one side, we adopt the residual check idea of Online-ABFT for our algorithm. On the other side, we combine backward/forward recovery, influenced by the Online-Detection/Online-Correction work of [10]. Combinations of redundancy and checkpointing have been favourably used in less application-specific contexts as well (e.g. [17]).

In summary, our contributions to the area of fault-tolerant iterative solvers are:

- We design and implement an original fault-tolerance PCG algorithm, TwinPCG. Its fault tolerance is rooted in using Dual Modular Redundancy
- TwinPCG detects faults, and implements efficient forward recovery from faults, with an intelligent detection and correction process; it can also perform rollback recovery, in the rare cases forward recovery is not deemed possible

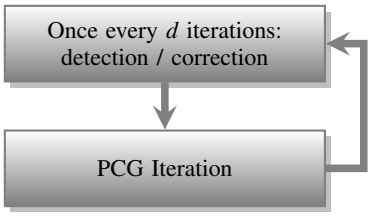

Fig. 1: Overview of fault-tolerant PCG implementations which do correction and detection on iteration level: Once every d iterations, faults are detected, and possibly corrected (through backward or forward recovery). At each iteration, textbook PCG step is performed.

- TwinPCG can utilize multi-threaded BLAS libraries like Intel MKL better than more wasteful solutions like Triple Modular Redundancy
- We implement from scratch our version of Online-ABFT. It has multiple optimizations: First, it uses parallel BLAS for SpMxV. Second, it reduces the sensitivity to insignificant faults, and as a consequence converges quicker than the original algorithm for the tested problems.
- We develop a flexible and realistic fault injection mechanism for all implemented solvers, and use it to confirm the fault tolerance of TwinPCG for a range of real-world problems

The rest of the paper is structured as follows: In section 2, we describe in detail our algorithm, TwinPCG. In section 3, we analyse the performance of TwinPCG, while in section 4, we evaluate its fault tolerance. We conclude the paper with section 5.

II. TwinPCG: Dual Modular Redundancy for PCG

A. Overview

In this section, we propose an original fault-tolerance algorithm for PCG, which we call TwinPCG. As shown in a high-level overview in Fig. 1, detection and correction are performed at PCG iteration level. Each PCG iteration is implemented in agreement with the reference algorithms listed in [18]. Our fault tolerance technique uses DMR to detect faults, and is capable of correcting faults as well. In most cases, the recovery is efficient forward recovery, as opposed to the more expensive checkpoint-restart recovery. In the very rare cases where forward recovery is not possible, we still perform a rollback to a checkpointed state.

We call our prototype TwinPCG for two reasons:

- First, our implementation uses two-threaded DMR. We consider this a minimal extension to single-threaded redundancy, and a less expensive technique than TRM (or any further thread redundancy).
- Second, both threads, much like twins, perform identical PCG iterations (each of them perform the steps shown in Fig. 1 on replicas of the same data). Also, they are very supportive of each other: whenever exactly one of them has a severe fault, the healthy thread recovers the faulty one.

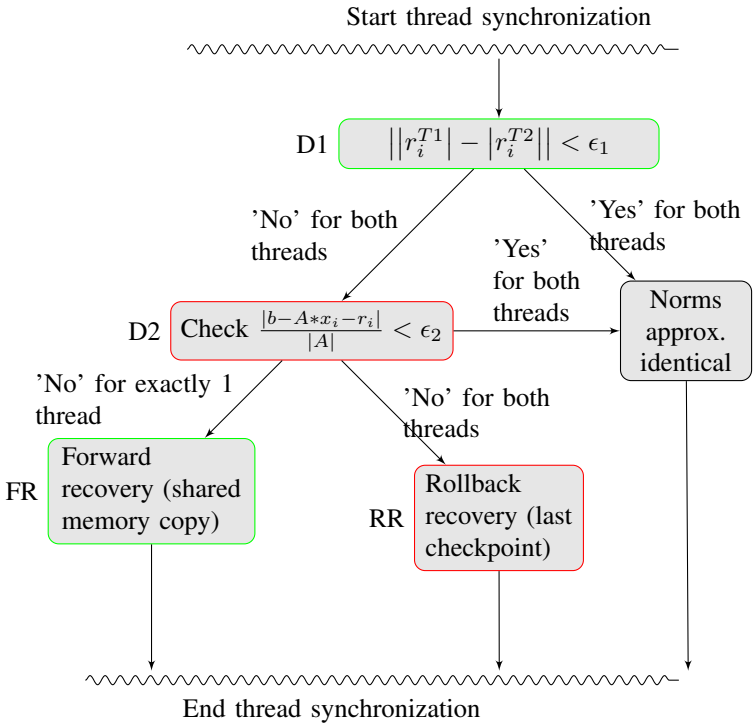

Fig. 2: Detection and correction logic of each TwinPCG thread. The efficient detection step D1, and efficient forward recovery FR, are marked in green. Both are only possible due to redundant multithreading. Their more expensive counterparts are marked in red – the detection step D2, and the rollback recovery step RR. We apply D2 and RR as our variations on Online-ABFT.

For some stages, steps from Online-ABFT [7] will be used and properly attributed. Due to space constraints, details of this related work are omitted.

In the following sections, we carefully describe the detection and correction phases of TwinPCG, the logic flow of these phases, and our reasoning behind the design.

B. Detection and Correction

In this part, we detail our implementation of detection and correction of faults for PCG, which we display in Fig. 2.

As shown in the diagram, the detection/correction phase, which we perform every d iterations, is enclosed by a synchronization window (at the start and at the end of each phase).

The synchronization window is essentially a software-based, and thereof more flexible version of a lock-step execution (see e.g. [19], Ch. 5.4.3).

Our synchronization window is needed for the following reasons:

- The start synchronization point is needed to ensure threads are in the exact same iteration in order to detect faults correctly
- The end synchronization point is needed, since the two threads need to read from shared memory (detection steps

D1 and D2), and to write into shared memory (correction steps FR and RR); we can not simultaneously allow another thread to perform a write operation as part of continued PCG iteration

As we see later in this work, the introduction of lock stepping in itself does indeed introduce a few percent overhead. Still, in the detection and correction of faults, we make savings in time, especially compared to checkpoint-restart mechanisms.

In the detection/correction phase, we use the redundant data at each thread. As a first detection mechanism (D1), we check the residual norm at each thread. In the absence of faults, the ideal case is $\left|r_i^{T1}\right| = \left|r_i^{T2}\right|$. Instead of checking for strict equality, we check that the difference is below a threshold ϵ_1. Apart from numerical reasons, this threshold has another very important implication – it becomes a significance filter for faults, as visualized in Fig. 3.

- If the inequality in D1 holds (always for both threads), either no faults have occurred, or faults have occurred that we consider insignificant. In other words, our use of residual norm in D1 allows us to use ϵ_1 as a significance filter for bit flips which may very often be insignificant. A wide range of bit flips which are insignificant in practice have been also observed in related work [20], [21]. Our solution has the implication that the two threads may diverge up to a certain point. No correction is needed until then.
- If this inequality does not hold (always for both threads), we only know that a significant transient fault has hit at least one, possibly both threads. Since we use dual redundancy, we can not use trivial mechanisms like majority vote to decide on where the fault occurs. For this reason, we rely on an invariant used in Online-ABFT. We check in parallel at each thread if the condition $r_i = b - A * x_i$ is fulfilled, and denote this detection step as D2. It requires an expensive matrix-vector product, but is only evaluated if detection D1 indicates an issue.
  - If the inequality holds at both threads, we continue without recovery
  - If one thread breaks the inequality, the healthy thread recovers the faulty thread via a trivial copy of its PCG data in shared memory. This forward recovery is efficient in many ways: it avoids reading a checkpoint (possibly from file), it avoids recomputation, and also importantly, no rolling back to previous iterations of PCG is required.
  - If both threads break the inequality, we assume that they both need to be recovered. We resort to a traditional checkpoint-restart, and we roll back a few iterations as Online-ABFT would do. We will later show that this RR (rollback recovery) step is extremely rare in practice for our solution.

The reader may express concerns over how ϵ_1 and ϵ_2 are chosen and calibrated. In our experiments in Sect. IV, we choose fixed threshold values which work well across all 8

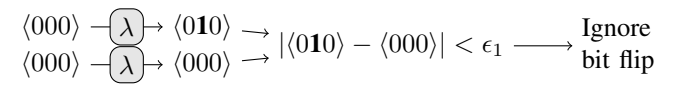

Fig. 3: D1 is both a detection mechanism, and filter for insignificant faults: Fault injection with probability λ for each thread, and threshold ϵ_1.

problems. Their values are indeed very important, and could be further optimized; the balance of the threshold level is in itself a complex topic. Furthermore, we refrain from establishing a relation between D1 and D2 in this work. For example, it is conceivable for a D1 check to fail, but a D2 check to pass, or vice versa, if the values are not properly set.

The underlying principle for D1 and D2 checks, regardless of calibration of thresholds, is solid in our view: a deviation between redundant threads indicates a soft fault (D1), and so does an invariant which does not hold any more (D2). We also remark that by design, detection steps D1 and D2 always provide consistent results across both threads, since they access shared data in a synchronized fashion.

C. Implementation Details

We implemented TwinPCG as a command-line tool in C. We use the reference algorithm of [18] to implement PCG, and the simple Jacobi matrix as a preconditioner. We use an implementation of File I/O and various operations for sparse matrices in the Compressed Row Storage (CSR) format [22]. There are the following external library dependencies:

- We use Intel Math Kernel Library for all Sparse BLAS operations (including OpenMP-parallel SpMxV routines)
- We use GSL [23] for the uniform probability distributions, and for the Poisson distribution; we use both for our fault injection mechanism, which we detail later
- We use POSIX threads to implement redundant multithreading, and POSIX condition variables and mutexes to implement the synchronization window

The entire implementation thus has no strict requirements on proprietary software. Intel MKL, as the only proprietary component, can potentially be replaced by an open-source Sparse BLAS implementation.

D. Fault Injection

We assume that data values of the problem matrix, and consequently any vectors, can be affected by transient faults. Since we use the efficient CSR format with its 3 arrays for values, columns, and rows, our assumption means we only allow the values array to be corrupted. While it is not complex to also design resilience mechanisms for the column and row arrays, this is not the subject of this work.

We have implemented from scratch a fault injection mechanism on the principle of fully uniform probability distribution, based on the assumption that any bit in memory is equally likely to get flipped. We assume that a bit flip can occur at any bit of any element of the value array of matrix A. It can easily be demonstrated based on a faulty matrix $\widetilde{A}$ in any given

iteration that its faults propagate to all PCG vectors irreversibly within one iteration. At the end of each iteration, we unflip any bit flips we may have introduced to the matrix, i.e. we fix the matrix A. This corresponds to the model of transient faults for PCG methods as used in [5], [10].

The fault rate λ corresponds to the mean rate of faults for a Poisson distribution; we generate faults accordingly. If a fault occurs in an iteration, we use uniform distribution to decide which element of the non-zero elements of A gets a bit flip. We then use again uniform distribution to decide which bit of this element gets flipped.

One consequence of this model is that often the injected faults have no noticeable effect on the convergence of PCG. This effect is well known, and there is relevant work [20], [21] studying how significant is a bit flip in a floating point number, depending on its position in the data representation. The mantissa of a Binary64 IEEE floating point number holds 52 bits; however, bit flips in the mantissa may often have no significant effect, especially if the exponent is very small. A fault injection framework using a uniform distribution of bit flips should take this into consideration. Indeed, in our experimental results, only a fraction of the injected faults can be considered significant (see Fig. 3).

III. Performance Evaluation of TwinPCG

In this section, we evaluate the performance of TwinPCG in following ways:

- We measure the memory requirements
- We carefully describe the thread-to-core pinning we use
- We compare our two-threaded implementation to a PCG implementation before memory contention due to SpMxV multithreading
- We evaluate the effects of enabling SpMxV multithreading in Intel MKL on PCG, the modified Online-ABFT, TwinPCG, and TMR

A. Experimental Platform

For our benchmarks, we use compute nodes on the Kelvin cluster at Queen's. Each node has 2-socket Intel machines with an Intel Xeon CPU E5-2660 v3 with 2.6 GHz frequency, and 128 GB RAM. Each of the two sockets has 10 cores, which share per socket 25 MB L3 cache, with 256 KB L2 cache, and 32 KB L1 cache per core. Intel Hyper-Threading was disabled on the nodes, so there are 20 physical and virtual cores available per node.

B. Main Memory Footprint

For its backward/forward recovery strategy, TwinPCG requires triple storage for all PCG vectors. A transient fault can permanently affect all PCG vectors within an iteration. Each of the two threads stores the PCG vectors (used for forward recovery), and a third copy is held in shared memory for backward recovery. Whether or not the problem matrix needs to be duplicated depends on the assumption – a transient or a permanent soft faults. For the transient fault assumption of this work, we only store a single copy of the matrix, and our fault injection mechanism restores the matrix at the end of an iteration to simulate this behaviour. We remark that redundant multithreading is so powerful that it allows for a trivial extension of our work to permanent soft faults. This case requires three copies of the input matrix, similar to the current PCG vector case, and trivial modifications to the existing code.

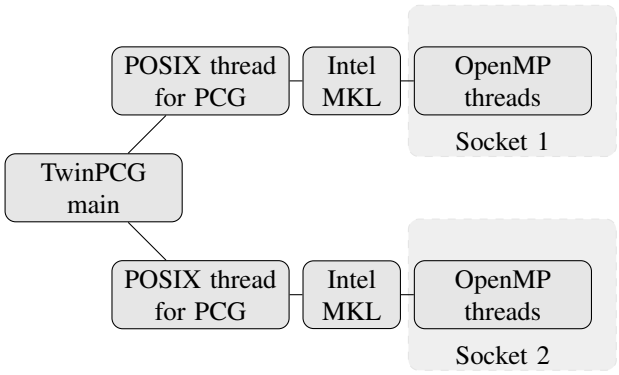

Fig. 4: Illustration of nested thread paralellism in TwinPCG, with each redundant POSIX thread calling the OpenMP-parallel SpMxV routine in Intel MKL. We also outline the thread-to-socket pinning we experimentally find to be most efficient.

We use the efficient Compressed Row Storage format for all matrices. The amount of storage needed is proportional to the number of nonzero elements (nnz). We profiled the heap use for various problems. We experimented with the range of matrices we use for most experiments, as listed in Table II, where the nnz varies between 340 thousand and 7.6 million. For our largest test matrix (G3_circuit with 7.6 million nonzero elements) and all vectors, our implementation consistently used 488.5 MB of heap memory with PCG, and 727 MB of heap memory for the two-threaded TwinPCG. This includes the entire allocated memory for all data structures. We therefore consider the problem of allocating a few extra hundred MB of additional memory per thread acceptable.

C. Efficient Thread-to-Core Mapping in TwinPCG

The thread-to-core mapping in TwinPCG is very important, and a bad mapping has a detrimental effect on performance, especially for a memory-bound problem like PCG.

In our implementation, there is a two level hierarchy of threads, as shown in Fig. 4:

- The main program first spawns POSIX threads (1 for PCG or Online-ABFT, 2 for TwinPCG, or 3 for Triple Modular Redundancy).
- Each PCG kernel calls an OpenMP-parallel SpMxV routine (implemented in Intel MKL)

While Intel MKL is thread-safe for multithreaded applications to use, the thread-to-core mapping when using POSIX threads to call OpenMP threads is not trivial. The challenges and solutions are described in detail in [24]. We are unable to properly set the thread affinity without source code modifications. The reason is that the OpenMP runtime only has a global

affinity view, but no notion of the POSIX thread which runs an OpenMP-parallel region. However, pinning down threads depending both on the first-level POSIX thread number, and second-level OpenMP thread number, is possible with some code modifications. The affinity can be set e.g. via Intel or POSIX Thread Affinity API.

The thread-to-core mapping for TwinPCG we choose fits very well with the used 2-socket platform: We place each POSIX thread, and all its associated OpenMP threads, on its own dedicated socket (the OpenMP thread groups are thus pinned to two different sockets, as shown in Fig. 4). The motivation is to avoid memory contention between the two redundant threads, which mostly use independent data sets.

For the TMR version, we use the same strategy for the first two threads, and pin the OpenMP threads of the third POSIX thread consecutively to sockets one and two. An ideal solution for TMR in this setting does not exist: we have two sockets available, and need to utilize three redundant PCG threads, each of them using a OpenMP-parallel SpMxV. Even in the presence of many-socket platforms, the fundamental problem of a fixed set of available resources would remain: kernels allowing for maximum concurrency would utilize these resources best, DMR solutions would be less efficient, and TMR solutions (or higher redundancy solutions) would be trailing off further in performance.

D. CPU Footprint

1) Evaluation of Multithreading and Synchronization in TwinPCG: It is important to evaluate the effects of multithreading in our two-threaded implementation of PCG. Two important questions arise:

- Is there a scenario where no memory contention occurs, and the efficiency of TwinPCG in wall-clock time is not affected compared to single-threaded PCG?
- For this scenario, how significant is the overhead of the synchronization window we presented?

To begin with, we experimentally confirm that memory contention can be avoided with following setting: use of a single-threaded Intel MKL SpMxV, and pinning of the two redundant POSIX threads, and the derived OpenMP threads, on different sockets to avoid competition for L3 cache.

After performing this setup, we run benchmarks with a number of matrix sizes, ranging from 340K to 7.7M nonzero elements. No faults are injected for both cases. Each run only terminates after convergence ($tol = 10^{-10}$). The average time over 10 iterations of each experiment was used for PCG and TwinPCG.

Fig. 5 shows the (average) PCG runtime for all problems, and the marginal TwinPCG overhead in runtime is indicated above the bars. PCG includes standard iteration time, while TwinPCG includes both iteration time and the synchronized detection (even though no faults are injected). When using proper thread-to-core pinning as described previously, the actual PCG computation time, excluding the lock stepping overhead of TwinPCG, is comparable across both single-threaded and two-threaded versions. TwinPCG did not introduce a slowdown in computation, even for the largest problems. However, the two TwinPCG threads did usually slightly differ in their compute time; for each lock stepping (once every 5 iterations), this translates into a forced slow down for the quicker thread. We profiled the maximum accumulated time in lock stepping across both threads for TwinPCG, and found that it amounts to at most 5.5%, compared to compute time. This difference translated into a TwinPCG overhead compared to PCG. It is an open question if this TwinPCG overhead can be eliminated; it may be due to inefficiencies in our lock stepping implementation, or a CPU/cache utilization difference for redundant computation on different sockets.

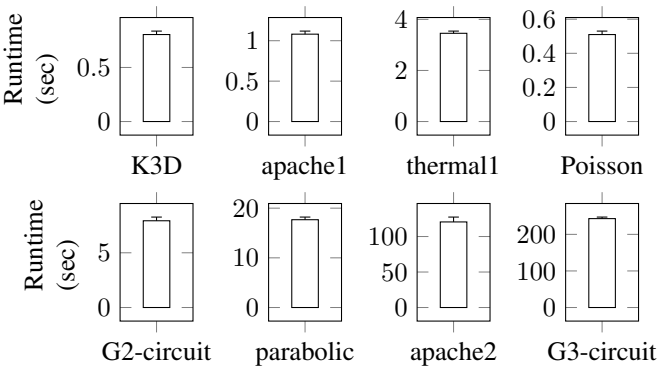

Fig. 5: TwinPCG runtime overhead: y-axis measures the average runtime to convergence for PCG for all problems. TwinPCG doubles the used threads, but we see at most 5-6% runtime overhead in runtime (overhead indicated above bars), which we attribute to lock stepping (Fig. 2) across the two redundant threads.

2) Combining Redundant Multithreading and BLAS Multithreading: Redundant multithreading for TwinPCG, and an efficient multithreaded SpMxV kernel as we use it, poses a conflict for resources for orthogonal purposes: fault tolerance, and efficiency. This conflict is intuitively clear, but its experimental validation is tricky, and requires precise pinning of threads to cores to avoid the significant effects of cache sharing for all tested settings. We have described our optimal choice of thread affinity, and we present experimental results based on this choice.

We experiment with our entire suite of implemented solvers, and a broad range of OpenMP threads for SpMxV parallelism. The results are shown in Fig. 6, the y-axis being the total computation time. More importantly, Fig. 6 shows the conflict between redundancy and BLAS efficiency. We have compared our solution TwinPCG with our reference implementation of PCG, Online-ABFT, and Triple Modular Redundancy. The shown problems are the largest two, apache2 (around 5M nnz), and G3_circuit (7.6M nnz); we choose larger problems for more representative results. The x axis represents the OpenMP threads we set *per PCG thread* for any implementation. For each kernel, we experiment with as many OpenMP threads per PCG thread as the total available cores; thus, we intentionally perform "stress tests" in order to properly evaluate the

overhead of redundant solutions. For PCG, and Online-ABFT, we naturally impose no thread-to-core restrictions, allowing the runtime to freely utilize both sockets. We expect PCG to exploit multithreading best, since no redundancy is used to "block" resources. We also expect Online-ABFT to exploit multithreading well, since its redundancy is sequential, and does not inhibit concurrency. Both of these expectations are confirmed: PCG and Online-ABFT perform well overall with increased BLAS multithreading. We then expect our two-threaded redundancy to come next, outperforming the TMR solution clearly for large OpenMP threads; this is due to both L3 cache contention, and ultimately oversubscription (for 10 and more OpenMP threads per PCG thread) for TMR. This expectation is also confirmed. We add here that our detection time (step D1) consistently outperformed Online-ABFT's D2, but this gain is generally lost in the overhead we incur through lock stepping.

In summary, the experimental results fully mirror the increasing level of redundancy as summarized in Table I. TwinPCG, while offering a much broader and generalized recovery spectrum than Online-ABFT, or any ABFT SpMxV approach, comes very close in performance to it. However, Online-ABFT does have an advantage for high levels of BLAS multithreading, since the thread redundancy of TwinPCG limits the available cores.

IV. Fault Tolerance of TwinPCG

We previously evaluated the performance of TwinPCG without faults, and without verifying in any way that the proposed scheme offers fault tolerance capabilities. In this section, we verify using a range of problems that in practice, TwinPCG recovers almost exclusively using the efficient forward recovery familiar from mechanisms like TMR.

A. Experimental Setup

In this section, we will evaluate the fault tolerance of TwinPCG in regard to transient faults. In the absence of transient faults, PCG, Online-ABFT, and TwinPCG converge in the same number of iterations. As we have outlined in section II, we expect to be able to perform efficient forward recovery for the case of a single-thread fault for the two redundant threads; In the unlikely event of a double fault (both threads experience a fault within a detection window), we perform backward recovery.

The parameters we use are:

- a detection is triggered each 5 iterations, and a backup is performed each 10 iterations (as in Online-ABFT)
- $e_1 = 10^{-15}$, $e_2 = 10^{-10}$ (See Fig. 2)
- We use $e = 10^{-10}$ for our version of Online-ABFT, for compatibility with the original, and in agreement with e_2 of TwinPCG
- We use as residue norm tolerance (termination criterion) 10^{-10} for all kernels.
- We abort each run if no convergence is reached at 6000 iterations

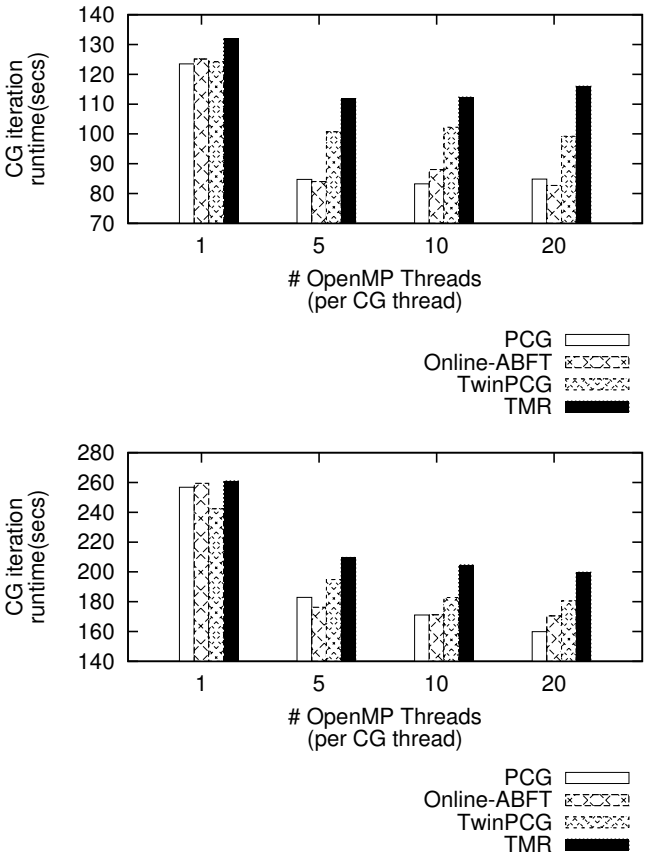

Fig. 6: Effect of using both redundant threads for fault tolerance, and multi-threaded SpMxV operations via MKL threads. Problems shown are apache2 (above) and G3_circuit (below). PCG and Online-ABFT use no concurrency, and therefore can better explore SpMxV multithreading, especially for larger MKL counts. TwinPCG can still explore multithreaded SpMxV with very little loss in performance (with efficient thread-to-core pinning); TMR experiences memory bottlenecks for larger OpenMP thread count (per PCG thread).

We should add that the detection (each 5 iterations), and the backup (each 10 iterations) are not necessarily optimal, and simply borrowed from the one proposed in the original Online-ABFT. Since our implementation is parallel, and hopefully more efficient, we intend to calibrate these intervals according to the widely accepted Young/Daly formula [25], [26].

The problems we use are all from the freely available University of Florida Sparse Matrix Collection [27], with the exception of the better conditioned K3D problem [5]. We list the problems, their nnz count, and their condition number estimate (according to MATLAB) in Table II. Our main criteria was to have a good range of real-world problems, which converge quickly with the simple Jacobi preconditioner in place for a fault-free execution.

We can experiment with arbitrary fault injection rates. We denote the average number of faults per iteration with λ. We show results for $\lambda = 0.01$ (a mean of 1 random fault per

Problem	nnz	condition number
K3D	340200	645
apache1	542184	$4 * 10^6$
thermal1	574458	$5 * 10^5$
Pres_Poisson	715804	$3.2 * 10^6$
G2_circuit	726674	$2 * 10^7$
parabolic_fem	3674625	$2.1 * 10^5$
apache2	4817870	$5.3 * 10^6$
G3_circuit	7660826	$2.24 * 10^7$

TABLE II: Selection of problems used in our benchmarks and fault tolerance tests.

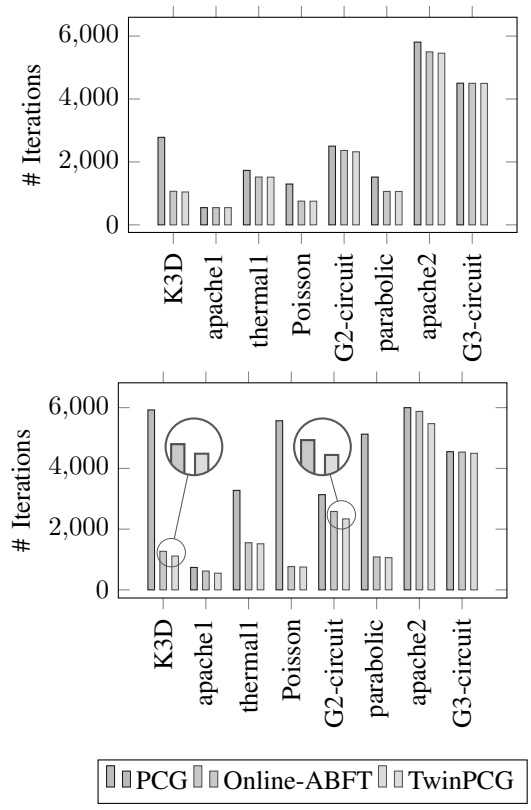

Fig. 7: Validation of fault tolerance of TwinPCG, with PCG and the modified Online-ABFT as reference. y-axis shows # iterations to converge. A run is terminated either at convergence, or at max 6000 iterations. Each result is an average over 60 iterations. Above: Fault rate $\lambda = 0.01$ (1 bit flip every 100 iterations). Below: Fault rate $\lambda = 0.1$ (1 bit flip every 10 iterations).

100 iterations) and $\lambda = 0.1$ (a mean of 1 random fault per 10 iterations). Some bit flips may be insignificant; similarly, some problems may not be ill-conditioned enough to be easily affected by bit flips. We tested all problems presented in Table II. Apart from our implementation TwinPCG, we experiment with our reference implementations of PCG, and our modified version of Online-ABFT; this version converges faster than the original in all our settings due to reduced fault sensitivity, and significantly less unnecessary rollbacks.

The compiled results are shown in Fig. 7. We show PCG (no recovery), our version of Online-ABFT (rollback recovery), and TwinPCG (mostly forward recovery). Every single bar is an average over 60 independent executions, each triggering different bit flip patterns for a probability λ.

The y axis shows the number of iterations to convergence (we abort at 6000 iterations at most per run). We consciously choose the number of iterations as a metric for fault tolerance rather than the overall runtime. This measure allows for a more educated comparison of TwinPCG against Online-ABFT, and their strengths and weaknesses. Fig. 7, especially for the higher fault rate, shows that the problems K3D, apache2, Pres_Poisson, and parabolic_fem are most sensitive to bit flips; the reasoning behind this is difficult, and factors like conditioning and sparsity of each matrix play a role. PCG (no fault tolerance) rarely converges for $\lambda = 0.1$ for either of these problems. For the same fault rate, TwinPCG reliably recovers forward, and always converges in less iterations than Online-ABFT; for example, it needs on average 9% less iterations for apache1 and apache2, 12% less iterations for G2_circuit, and 13 % less iterations for K3D (highlighted in the figure below for K3D and G2_circuit). We stress that these are averages over 60 different iterations per setting (problem, error rate, and kernel). In addition, TwinPCG very rarely resorts to rollback recovery: the highest rollback recovery ratio to forward recovery was 1:60 (for the G2_circuit and $\lambda = 0.1$). This outcome makes it unnecessary to draw a comparison to the more resource-expensive TMR method, which in its forward recovery behaves as TwinPCG in practically all settings. The rollback recovery (via checkpoint) in TwinPCG is very rare due to the low probability of double faults before correction. This raises the idea of eliminating rollback recovery altogether for very low fault rates, and activating it only for higher fault rates, in future work.

Under the assumption that we use multithreaded BLAS on the available cores, TwinPCG can not always outperform Online-ABFT in total runtime; the overall runtime is faster for the 5 smallest (out of 8) problems. For the largest problems, as visible in Fig. 6, Online-ABFT explores multithreaded BLAS better. However, for the higher λ fault rate, TwinPCG runtime remains practically constant due to forward recovery, while Online-ABFT runtime increases (Fig. 7). If we extrapolate this result, a higher fault rate should always lead to an advantage for TwinPCG, even in the face of less cores available for BLAS multithreading. Still, for low fault rates less redundant solutions like Online-ABFT are better than more redundant solutions like TwinPCG.

V. Conclusion

In this work, we introduced a fault-tolerant PCG implementation, called TwinPCG, which uses two redundant threads, and is able to both detect and correct transient faults. The recovery is powerful, in the sense that it covers arbitrary faults within matrices and vectors in PCG, including SpMxV faults studied elsewhere. The transient fault assumption can also be trivially extended to any soft faults. In terms of CPU usage, our dual redundancy doubles the used CPU and cache resources compared to PCG, but it does not triple them as TMR.

We concluded that the introduced lock stepping across two threads adds up to 6% overhead relative to the baseline in our implementation, but no other overhead is introduced compared to a single-threaded PCG implementation, in terms of wall-clock time. To achieve this performance, we designed a proper thread-to-core pinning. TwinPCG exploits BLAS parallelism well, but as expected solutions like Online-ABFT can utilize more cores for parallelism. We tested the fault tolerance of TwinPCG on a number of problems, and concluded that its ability to recover from faults shows all the strengths that TMR solutions have, almost exclusively performing forward recovery and outperforming therefore algorithms like Online-ABFT for ill-conditioned problems and increasing fault rates.

In summary, we see TwinPCG as a more efficient, and equally robust, alternative to TMR solutions for PCG, with the same powerful recovery capabilities. TwinPCG is particularly attractive for settings where higher fault rates may be anticipated, in particular near-threshold voltage settings of the future.

ACKNOWLEDGEMENT

This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 671603.

We would like to thank Enrique Quintana Ortí and José I. Aliaga of Universitat Jaume I for providing an initial implementation of CG with Intel MKL and CSR format support, and for answering all related questions. We are also thankful to Giorgis Georgakoudis and Charalampos Chalios of Queen's University Belfast for the many insights into cache hierarchies, thread parallelism, and runtimes.

REFERENCES

[1] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*. NBS, 1952, vol. 49.

[2] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: an architectural perspective," in *11th International Symposium on High-Performance Computer Architecture*, Feb 2005, pp. 243–247.

[3] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-threshold computing: Reclaiming Moore's law through energy efficient integrated circuits," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 253–266, 2010.

[4] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar, "Near-threshold voltage (ntv) design: Opportunities and challenges," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1153–1158. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228572

[5] P. Sao and R. Vuduc, "Self-stabilizing iterative solvers," in *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. ACM, 2013, p. 4.

[6] J. Langou, Z. Chen, G. Bosilca, and J. Dongarra, "Recovery patterns for iterative methods in a parallel unstable environment," *SIAM Journal on Scientific Computing*, vol. 30, no. 1, pp. 102–116, 2008. [Online]. Available: http://dx.doi.org/10.1137/040620394

[7] Z. Chen, "Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods," in *ACM SIGPLAN Notices*, vol. 48, no. 8. ACM, 2013, pp. 167–176.

[8] L. Jaulmes, M. Casas, M. Moretó, E. Ayguadé, J. Labarta, and M. Valero, "Exploiting asynchrony from exact forward recovery for due in iterative solvers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 53:1–53:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807599

[9] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Fault tolerant preconditioned conjugate gradient for sparse linear system solution," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 69–78. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304588

[10] M. Fasi, Y. Robert, and B. Uar, "Combining backward and forward recovery to cope with silent errors in iterative solvers," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, May 2015, pp. 980–989.

[11] S. Hukerikar, K. Teranishi, P. C. Diniz, and R. F. Lucas, "An evaluation of lazy fault detection based on adaptive redundant multithreading," in *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, Sept 2014, pp. 1–6.

[12] Z. Chen, "Algorithm-based recovery for iterative methods without checkpointing," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC '11. New York, NY, USA: ACM, 2011, pp. 73–84. [Online]. Available: http://doi.acm.org/10.1145/1996130.1996142

[13] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *Computers, IEEE Transactions on*, vol. 100, no. 6, pp. 518–528, 1984.

[14] V. P. Nelson, "Fault-tolerant computing: fundamental concepts," *Computer*, vol. 23, no. 7, pp. 19–25, July 1990.

[15] S. Hukerikar, P. C. Diniz, R. F. Lucas, and K. Teranishi, "Opportunistic application-level fault detection through adaptive redundant multithreading," in *High Performance Computing & Simulation (HPCS), 2014 International Conference on*. IEEE, 2014, pp. 243–250.

[16] P. C. Kanellakis and A. A. Shvartsman, *Fault-tolerant parallel computation*. Springer Science & Business Media, 2013, vol. 401.

[17] X. Ni, E. Meneses, N. Jain, and L. V. Kalé, "Acr: Automatic checkpoint/restart for soft and hard error protection," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 7.

[18] Y. Saad, *Iterative methods for sparse linear systems*. Siam, 2003.

[19] S. Poledna, *Fault-tolerant real-time systems: The problem of replica determinism*. Springer Science & Business Media, 1996, vol. 345.

[20] J. Elliott, F. Mueller, M. Stoyanov, and C. Webster, "Quantifying the impact of single bit flips on floating point arithmetic," *preprint*, 2013.

[21] P. Gschwandtner, C. Chalios, D. S. Nikolopoulos, H. Vandierendonck, and T. Fahringer, "On the potential of significance-driven execution for energy-aware hpc," *Computer Science-Research and Development*, vol. 30, no. 2, pp. 197–206, 2015.

[22] "Netlib - compressed row storage," http://www.netlib.org/utk/people/JackDongarra/etemplates/node373.html, accessed: 2016-04-29.

[23] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi, "Gnu scientific library reference manual-(v1. 12)," *Network Theory Ltd*, vol. 83, 2009.

[24] "Using threaded intel mkl in multi-thread application," https://software.intel.com/en-us/articles/using-threaded-intel-mkl-in-multi-thread-application, accessed: 2016-04-29.

[25] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Communications of the ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[26] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303–312, 2006.

[27] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.